DASICS-安全处理器设计白皮书

DASICS - Secure Processor Design White Paper

版本：v1.0.0

中国科学院计算技术研究所
先进计算机系统实验室系统结构组

2023 年 4 月 10 日

目录

1 简介

开源、共享和协同的软件开发模式促进了互联网、人工智能等领域繁荣发展，但在这种模式下软件开发的复杂性日益增加，体现在依靠大量开发者共同开发一个软件、频繁调用第三方代码库以及管理维护庞大的整体代码量。这种复杂的软件开发模式导致了在开发层面很大概率会引入安全漏洞。例如软件开发者不可避免地需要调用第三方代码库，却缺乏对第三方代码库的安全性的保证，导致了由于调用不可靠的第三方代码库引入了可以被攻击者利用的漏洞，带来信息泄漏和篡改的风险。一旦一个经常使用的第三发库发现漏洞，受影响的往往是大量使用这个库开发的软件。

软件安全漏洞中最主要是内存访问漏洞。针对这些内存访问漏洞带来的软件安全挑战，学术界和工业界提出了一系列软硬件内存防护方法。这些防护方法一方面通过数据流完整性技术（Data Flow Integrity，简称 DFI），对非可信的软件代码的数据流进行严格的检查和限制，通过对数据边界的越界检查或者数据来源的合规性检查等来防止对内存的非法操作。这其中代表性的工作包括工业界中 Intel 公司提出的 MPX 和 MPK 技术、ARM 公司的 MTE 技术以及英国剑桥大学主导的 CHERI 安全体系结构等。另一方面通过控制流完整性技术（Control Flow Integrity，简称 CFI）来防止恶意的控制流劫持，例如 Intel 的 CET 技术、ARM 公司的 BTI 技术和 PA 等技术。但是这些内存防护方法不同程度地存在着隔离划分对象粒度过粗、安全元数据容易遭受攻击或者硬件实现/性能开销过大以及需要对现有第三方代码进行大幅修改和重新编译的问题。

我们提出了 DASICS 安全处理器设计方案，以解决现有安全防护技术的隔离对象粒度过粗、元数据安全性低、性能开销过大的问题，并关注先前工作较少关注的权限动态划分、同一级地址空间内的内存保护和跨层调用检查。实现一种基于代码片段做权限动态划分的安全处理器设计，提供硬件辅助的高效软件内存防护，保障第三方代码的安全调用和运行，为基于开源开放的软件开发提供坚实的安全保障和支撑。

2 不可信的第三方代码库

如今许多底层软件（例如操作系统、网络协议栈等）、高性能第三方库（例如 SQLite、ASL、Crypto++、Qt 等）都基于非内存安全的 C/C++ 语言开发。其中有不少第三方库都存在潜在的内存访问漏洞。比如曾经发现过的存在堆缓冲区溢出漏洞的 XML 解析库 libxml2、有远程代码执行漏洞的 HTML 代码过滤库 htmlawed 以及有缓冲区溢出漏洞的开源反恶意软件工具包 ClamAV。软件开发者在使用这些第三方库时，往往忽略了这些库代码本身的一些非安全特性（例如不对程序的输入进行边界检查等），而导致基于这些库开发的软件存在可以被利用的安全漏洞，造成信息泄漏和篡改等严重的软件安全问题。

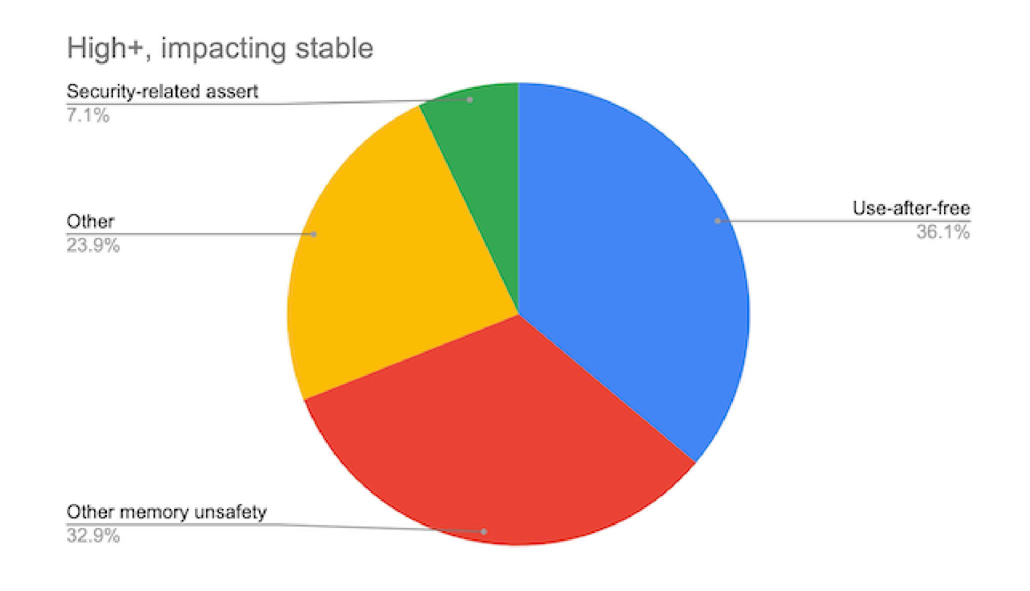


图 1: Chrome 浏览器安全漏洞统计

内存访问漏洞是当下软件安全问题的主要威胁。所谓的内存访问漏洞是指程序在对内存进行操作时，由于编程错误或恶意攻击，导致访问或修改了未被授权的内存区域，从而可能破坏程序的正常运行，甚至导致系统崩溃或被攻击者利用。常见的内存访问漏洞包括缓冲区溢出、使用未初始化的内存、空指针引用、堆溢出等等。这些漏洞可能导致程序崩溃、数据丢失、系统崩溃、信息泄漏等安全问题，给系统带来极大的风险。图 1 给出了 Google 统计的 chrome 谷歌代码库中较难修复的安全漏洞的来源，统计显示其中 36.1% 的漏洞威胁来自于 Use-after-free 攻击，主要是由不正确的内存管理造成的内存空间释放后再利用而导致的漏洞；32.9% 来源于溢出攻击等其他安全问题。从统计的安全漏洞上看，大概有 96% 的漏洞是内存访问漏洞。

第三方库代码里典型的内存访问漏洞产生的主要原因，是没有对非可信的代码进行可访问数据的边界检查，以及没有对程序的控制流进行检查。例如加密程序库 OpenSSL 1.0.1 版本之前出现的 Heartbleed 漏洞（CVE-2014-0160），该库在传输层（TLS）协议中实现了心跳检测（Heartbeat）机制时，没有对输入的长度进行验证（即缺少边界检查），导致攻击者可以利用这个漏洞对客户/服务器进行 Overread Buffers 攻击，造成关键信息的泄漏，影响范围广泛。对一些更为严重的内存漏洞，攻击程序可以利用漏洞进行指令执行地址的改写，使得程序的控制流转移到攻击者控制代码上，从而可以进行进一步更灵活和更具威胁性的攻击，甚至具有了图灵完备性的能力。

通常第三方代码库和软件开发者自研的代码运行在同一个进程地址空间内，因此第三方代码的漏洞很容易影响到整体程序的安全。对于用 C/C++ 等不内置内存安全检查的语言编写的程序，理论上每一行代码都可以不受限制的直接访问进程地址空间内的任何位置。一旦第三方代码的漏洞被攻击者利用，就可以很容易的访问到进程地址空间内的任意数据和代码，窃取或修改数据，甚至劫持程序并进一步进行操作系统提权尝试。而操作系统内核里的第三方驱动的漏洞一旦被利用，则可以访问到整个系统的内存。

现有的针对内存访问漏洞防护的研究工作大致分为两个方向：一方面从数据流的角度关注于如何高效地对软件进行区域划分并在划分基础上进行边界检查。区域划分和边界检查限制了某些代码只能访问自己的数据不能越界访问其他代码的数据，比如函数 A 不能通过栈溢出修改上一级函数的局部变量。另一方面从控制流的角度进行跳转限制，比如函数 A 返回的时候如果返回地址不是原来调用后的地址就需要触发例外防止不合法的跳转。不同的安全防护方法在

完备性、易用性和性能开销等方面具有不同的特点，在实际的系统设计选择时需要进行仔细权衡。

3 相关工作对比

针对内存安全问题的防御机制可以分类为纯软件方法和软硬件协同的方法。纯软件的方法可以从编程语言、编译器以及加载器等不同的层面进行防护。而软硬件协同的方法包括构造程序运行的可信执行环境、以及在程序内部进行的地址空间隔离和控制流保护等方法。

3.1 纯软件方法

纯软件的防御方法可以从各个层面进行防护。编程语言方面，如果采用例如 Rust 等内存安全语言对非可信代码进行重写，可以解决大部分内存访问漏洞问题。然而这种方法面临着极大的实用性困难，因为种类繁多、代码庞杂的第三方库都进行从 C/C++ 到 Rust 等安全语言的移植无疑是一个工程量巨大的工作。

编译器方面，通过对源代码插入一些实时检查代码，可以防护运行时的越界行为。例如 GCC 和 clang 上已实现的 Stack Protector 技术，主要通过在函数调用时设置一个栈保护变量，并在函数返回时检查其是否被篡改来防止针对栈的攻击。另一种是微软 C/C++ 编译器已经开始支持的 AddressSanitizer(ASan) 编译器插件，它可以在编译时将检测代码注入到目标程序中，并在运行时检测内存错误。

在程序加载时，主要使用地址空间随机化技术（Address Space Layout Randomization，简称 ASLR）进行防御。ASLR 是是一种针对利用内存漏洞进行代码跳转攻击的安全保护技术，其基本思想是通过对堆、栈、共享库映射等线性区布局的随机化，让攻击程序无法找到确切目标代码位置，从而增加攻击者预测目的地址的难度。

软件对于控制流的保护主要依赖 CFI 技术，该技术首先通过静态分析程序源码或二进制码分析刻画程序的控制流图 (control-flow-graph, CFG)，并在程序运行时通过监测程序控制流是否始终处于 CFG 中来判断程序是否遭受攻击。CFG 是一个图，由基本块和有向边组成。每个基本块代表一段连续代码，其中不包含控制流跳转指令和其他控制流跳转指令的目标。每条有向边代表程序控制流的一次转移。

纯软件方法虽然已经能够提供一定的安全防御，但是由于需要进行代码插桩（例如 ASan 等方法）等方式，在访存前后添加检查代码导致显著降低了原本软件运行的性能。同时需要对安全元数据自身进行保护的机制（例如 CFG 检查控制流方法）也必须保证这些用于安全判断的安全元数据不会被攻击者篡改和伪造。

3.2 可信执行环境

可信执行环境（Trusted Execution Environment，TEE）是一种基于软硬件协同的安全架构，它通过时分复用 CPU 或者划分部分内存地址作为安全空间，构建出与外部隔离的安全计算环境，并通过硬件隔离、加密或者完整性验证保障恶意程序无法从环境外部读取和篡改内部的数据和控制流，从而提高系统的安全性和私密性。TEE 在手机、云计算等领域得到了广泛应用。这其中的代表性技术是 Intel 的 SGX 安全技术、AMD 公司的 SEV-SNP 安全虚拟化技术技术、ARM 架构上的的 CCA 机密计算技术和 Trustone 处理器安全隔离技术、RISC-V 架构上的 Penglai 和 Keystone 等。

特定的 TEE 技术需要依赖于特定的体系结构或者平台，增加了设备更换和软件移植的成本。同时 TEE 为了维护计算环境的安全性，需要设置大量的硬件资源（比如用默克尔树等方式保障内存数据完整性）来完成保护。通常，TEE 内部的程序和 TEE 外部的数据交互的开销是比较大的，同时安全环境的创建和切换的性能开销也不容忽视。因此 TEE 适合运行相对独立的小规模的安全核心代码，而不太适合需要频繁交互的场景。理论上，把一个大型的应用程序完全运行于 TEE 内部也是可行的，但是大型程序内部多个模块之间的隔离，特别是对第三方库的隔离是 TEE 功能没有考虑的。

3.3 地址空间安全隔离

除了可信执行环境（TEE）等针对软件外部提供的隔离保护之外，还需要针对软件内部的威胁（例如非可信的第三方代码库）进行防护。假设软件内部没有进行代码之间的隔离。所有代码都是相同的权限，任意代码都可以访问地址空间内的任意数据并跳转到同一地址空间内的任意其他代码，那么一些非可信的代码将会以此对软件处理的私密数据（例如用户密码）等进行窃取。甚至还会跳转到恶意的攻击代码进行更具威胁性的攻击。因此软件内部也需要根据不同的代码进行权限的划分和访问以及跳转区域的隔离。

软件内部防护技术需要对软件内部的代码片段进行权限划分，同时对非可信的第三方库需要进行访问内存的限制和跳转目标的限制，以防止越界访存篡改私密数据，同时防止控制流劫持跳转到恶意代码进行篡改。因此可以从数据访问隔离和控制流保护两个方面对软件内部的防护技术进行分类。

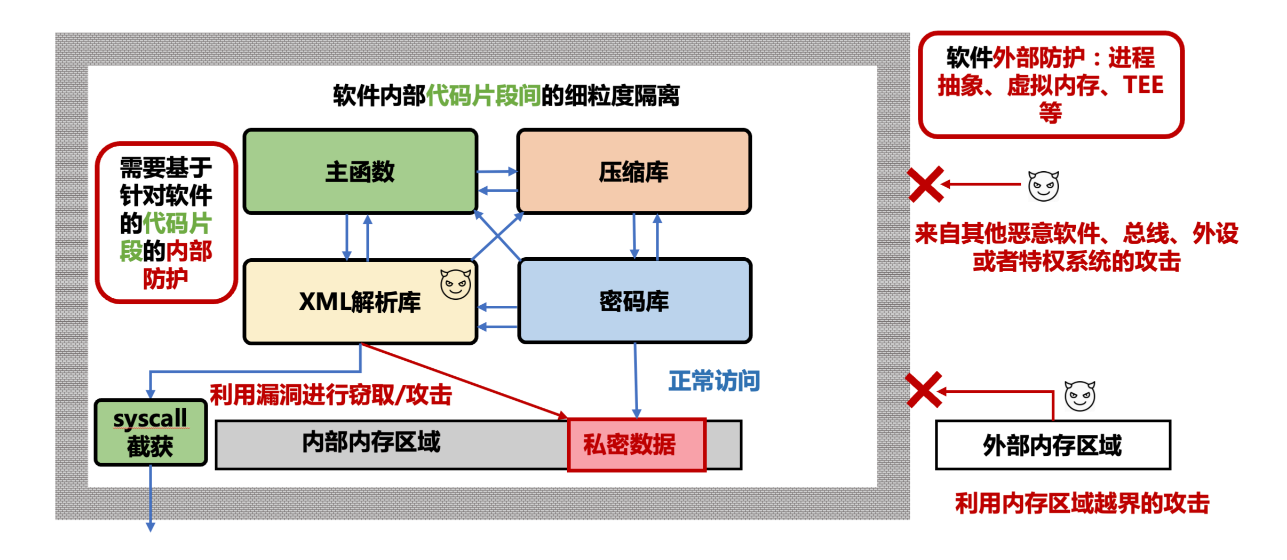


图 2: 来自软件内部非可信库的威胁

3.3.1 DFI 方法

数据流完整性技术（Data Flow Integrity，简称 DFI）的思路主要是通过对软件内部不同代码允许访问的数据进行划分和隔离，同时在进行访存时根据这个隔离的规则对访存进行检查，防止越界访问和非法操作，以阻止对敏感数据等关键信息的窃取和篡改。根据划分的方式不同，DFI 方法还可以进一步分为针对指针附加权限和针对内存数据附加权限。

对指针附加权限的代表性技术包括 MPX 和 CHERI 等，主要思想是规定每个指针访问的范围以及对指针指向的数据所允许的操作，在使用指针进行访存时根据这些规定进行检查。

MPX（Memory Protection Extensions）是 Intel 公司开发的一种用来防止内存访问漏洞硬件扩展技术。MPX 增加了 4 个 128 位的界限寄存器，4 条限界地址设置和比较指令，4 条限界寄存器的移动指令，两个配置寄存器和 1 个状态寄存器。在程序执行时，MPX 通过检查指针引用，判断对该指针的引用是否超出了预设的范围，从而提高代码的安全性。因为限界寄存器有限，所以更多的指针限界元数据必须另存放在内存的一个表中。使用这种技术由于需要对第三方库都进行重新修改编译，软件修改代价很大，且软件对每一条访存指令进行代码插桩比较，带来严重的性能下降。众多的问题让 Intel 于 2019 将 MPX 技术从 x86 扩展中移除。

CHERI（Capability Hardware Enhanced RISC Instructions）是另外一种软硬件协同的内存防护技术。CHERI 技术通过扩展指针的语义，将原来地址长度的指针扩展为 128 位包含这个指针有效范围、权限、是否有效等信息的扩展指针 (Capability)。并添加专用的指令修改扩展指针信息。CHERI 可以实现指针级别的内存安全保护，理论上能够防御大多数溢出攻击。但是 CHERI 的扩展指针使得这项技术的访存开销十分巨大。同时它要求插入修改扩展指针的命令，导致需要编译器的大量修改以及第三方库代码的重新编译。

对数据附加权限的代表性技术包括 MTE 和 PKU 等，主要通过对内存数据区域进行标记（tag）进行数据隔离，代码通过持有这些标记来明确对对应内存区域的访问权限。

MTE (Memory Tagging Extension) 技术是 ARMv8.5-A 架构中引入一种内存保护机制，它使用 tag 将内存分成多个 64 字节的部分，并为每个部分分配一个唯一的 tag。这些 tag 与指针一起使用，以验证指针是否指向正确的内存位置。MTE 主要通过使用这些硬件 tag 来执行内存保护，而不需要对代码进行任何修改。此外，MTE 还提供了软件 tag 功能，以允许应用程序通过多种 tag 方案自定义 tag，包括基于地址的标记和基于随机数的标记。它还提供了多个标记宽度选项，以平衡标记的数量和标记带来的额外开销。

PKU (Protection Key Unit) 技术是一种轻量级的内存隔离机制。以 Intel 的 Memory Protection Key (简称 MPK) 技术为代表，依靠页表项多余出来的 4 位作为 key 值给页面“上色”的方式对内存区域按页粒度进行分区。MPK 添加

了一个特殊寄存器（PKRU）来记录当前程序的 key（“色号”），当被访问的页面不允许当前 key 访问时就会检测到非法访存。MPK 还添加了特殊指令来对 PKRU 进行操作。这样就用较少的硬件开销实现了内存隔离。但是 MPK 本身没有成体系的安全机制来保障 PKRU 不被篡改，因此需要对库函数的二进制进行检查确保库函数没有修改 PKRU 的指令。同时，MPK 最多支持 16 个 key，也就只能同时存在 16 种着色，这对有大量隔离需求的复杂程序来说是远远不够的。

3.3.2 CFI 方法

控制流完整性技术（Control Flow Integrity，简称 CFI) 原本是需要构造程序控制流的完整刻画（比如 CFG)，但在实际应用技术上，共性思路是采用简洁高效的方式保护控制流关键数据（比如函数指针、栈上的返回地址等)，以及针对间接跳转的目标地址进行限定（比如只能是函数入口等)。主要技术包括 CET、BTI 以及 PA 等

CET(Control-flow Enforcement Technology) 技术是英特尔（Intel）公司开发的一种新的安全技术，主要通过在硬件层面上保护控制流，可以有效地防止针对控制流进行劫持和篡改。CET 具体实现包括两种机制：

- Indirect Branch Tracking (IBT)：主要是针对 JOP (Jump-Oriented-Programming) 和 ROP (Return-Oriented-Programming）等利用间接跳转的攻击。它通过为间接跳转指令设置着陆点指令来限定程序中合法的间接跳转目标地址（比如函数入口地址)，以确保跳转目标的合法性。

- Shadow Stack (SS)，它通过维护一个额外的栈来跟踪函数调用的返回地址：在函数调用时把返回地址进行入栈，在返回时对返回地址进行弹出同时和函数返回时候的跳转目标进行对比，以此来防止攻击者篡改返回地址。

BTI（Branch Target Identification）是 ARMv8.5 增加的一种安全特性，也是针对 JOP 这种攻击方式。和 IBT 类似，它在译时会为间接跳转指令设定特定位置对 BTI 指令，以此来规定间接跳转的目标，增加了 JOP 通过间接跳转将程序串联在一起进行攻击的困难。

PA（Pointer Authentication）技术是 ARM 公司一项指针完整性校验技术。PA 技术利用密码学完整性来保护返回地址，同时出于节省硬件的原则，PA 利用了虚地址指针中的空闲高位。在编译器的配合下，被调用函数首先计算出返回地址的校验码，并附在在指针高位中（称为 PAC，Pointer Authentication Code)，再将带校验码的返回地址压入栈中。然后在函数返回前插入校验指令。如果发现校验结果和 PAC 不一样，则说明返回地址被恶意篡改，继而触发异常。

3.3.3 安全元数据保护

现有的地址空间隔离方法存在着一个共性的问题：一些保护机制在使用安全元数据（比如 MTE 的 tag 和 PKU 的 key）进行防护的同时，却缺乏对安全元数据的保护，导致恶意程序可以通过修改安全元数据提升自己的权限或者取消隔离限制来使保护机制失效。例如针对 MPK 保护机制，程序可以使用指令随意操作 PKRU 寄存器使得针对自己的着色保护失效。

针对安全元数据保护问题，有大量的研究尝试从不同方面进行改进，大致可以分为两个方向：

- 二进制扫描方法：这种方法主要通过对应用程序或者非可信库的二进制代码进行检查和重写，保证非可信代码不能存在对安全元数据进行修改的指令（一般是一些特殊设计的指令)。例如学术界提出的 ERIM 方法通过二进制扫描防止第三方库代码包含对 PKRU 寄存器的修改指令 WRPKRU。但是这种方式对于一些动态生成的代码并不能支持。

- 用户态特权函数方法：这种方法的思想是在用户态中构造一个特权函数用来专门进行安全元数据的管理和操作，在这个函数之外进行的安全元数据操作都会触发例外。以学术界提出的 Donky 为例：Donky 使用用户态中断的方式在用户态设置一个安全管理函数 Domain Monitor，所有对 PKRU 的修改只能在这个函数中进行，在这个之外使用 WRPKRU 指令则会触发例外。同时 Donky 对于跳转到 Domain Monitor 也进行了控制流限制，防止程序通过控制流劫持进入 Domain Monitor 进行 PKRU 的改写。这种方式类似于对用户态进一步进行特权集划分，但即使是轻量级的特权级划分，也可能由于频繁的特权切换而带来的性能开销。

安全元数据的保护往往需要 DFI 和 CFI 结合的方式。首先这些元数据本身作为私密数据需要 DFI 的方法提供安全隔离，保证非可信代码无法访问和篡改。其次需要 CFI 的方法保证能够修改这些私密数据的代码不会被劫持和利用来进行恶意修改。

3.4 设计安全方法的共性问题

我们针对上述安全防护机制进行了总结，认为针对内存访问漏洞的安全机制在进入到实用阶段时，需要面临以下的实际问题：

3.4.1 防护能力和开销的权衡

有广泛应用场景的处理器安全方案不可避免的面临性能/硬件开销和防护能力之间的权衡。一方面，一些防护能力强的机制可能会由于频繁的检查带来巨大的性能开销（比如 MPX 技术需要针对每个指针访问都进行权限表的查询，使得它的性能开销非常大）。基于特权态防护进行的安全保护可能需要进行大量特权态切换，也会带来不小的性能开销。另一方面，一些软硬件协同的安全机制所依赖的额外的安全元数据可能会导致大量的硬件开销，比如 CHERI 对指针进行了扩展，指针变量的存储开销增加了数倍，因此对于大量存在指针的应用程序可能会导致不小的存储和访问开销。因此，在设计安全机制时不仅需要考虑安全性问题，还需要考虑达到这个安全性所需要的代价问题。

3.4.2 可移植性问题

一些安全机制在对现有的第三方代码库进行防护时，还存在着重写困难和编译困难，这导致现有安全机制的可移植性问题。许多学术工作通过设计添加新的指令或者扩展指针等方式来实现这一目标。然而这些工作更改了程序本身的运行指令和指针语义，导致和遗留的二进制代码库不兼容。如果要使用这些安全机制来做防护需要把所有非安全的第三方库重新进行编译，抛开工作量很大的问题，许多第三方库并不是开源的，因此许多安全机制很难广泛地进行应用。总之在设计安全机制时也需要考虑可移植性问题。

3.4.3 特权切换的安全检查

DFI 和 CFI 一般是在同一个特权级内部，增加细粒度的防护机制。但是，一旦进行特权级切换，高特权级的代码往往拥有对低特权级代码和数据的修改权限。因此，如果不对特权级切换进行一定的限制，那么 DFI 和 CFI 的保护机制可能通过特权调用的方式破坏。典型的特权级调用包括系统调用、虚拟机陷入等。其中系统调用是应用程序使用操作系统提供功能的重要接口，但也可以被非可信代码利用来绕过安全机制保护。比如在 MPK 保护机制下，代码可以利用 madvise 系统调用（原本是用来给内核更多访存信息来提高性能）来清除特定内存页中的内容，即使 MPK 保护这些页不会被修改。代码还可以通过 brk 和 sbrk 系统调用（原本是用来进行堆数据的管理）来回收和重新分配堆上的私密数据。

因此安全机制除了数据隔离、边界检查和控制流限定等功能外，还应该据有针对系统调用等特权级切换进行截获和过滤的功能，来防止利用特权级切换进行的攻击。

4 DASICS 设计思路

基于上述针对安全方法设计面临的问题，我们期望设计一种面向同一地址空间内的应用，兼顾数据流完整性、控制流完整性和系统调用安全性的高效率、低开销、可移植性好的处理器安全增强技术。

攻击模型： 我们的安全处理器设计所假设的攻击模型是软件内部可以分为具有不同来源的代码片段，其中包含了软件开发者所编写的部分，这部分代码可以由软件开发者所控制，也可以根据新需求重新编译。另外包括了第三方库代码部分，这部分代码可能来源于闭源代码生成的二进制代码或者动态生成的代码，开发者无法对这部分代码的源码进行分析和修改。我们将这部分代码假设为非可信代码，即很可能存在可以被利用的内存访问漏洞，甚至可能还包括通过系统调用等机制进行特权态提升的危险操作。攻击者可以利用这些漏洞进行控制流劫持、数据篡改以及关键数据的窃取等操作。

代码即主体: 要完成一个地址空间内部的安全隔离，首先面对的是如何区分权限主体 (subject) 的问题。在一个地址空间内部，传统的进程、线程不适合作为安全主体；单纯以函数库为主体粒度也太大；以指针或指令为主体粒度又过细。

我们注意到同一个程序内的代码片段（地址连续的一段代码）可以作为一个合适的权限划分的主体，例如一个库函数内部存在很多地址连续的代码片段，这些代码片段通常用来实现同一个功能。同一段代码在特定时间访问的数据范围是确定的、对数据的操作也是确定的，也就是说，可以对这段时间里的不同代码所操作的数据进行空间上的划分（比如限定指针的访问范围）以及操作上的限制（比如限制对特定地址的读/写）来达到特定时间的安全隔离。同时我们也需要随着时间迁移动态地来做划分和限制来保证同一代码片段在不同时间的安全隔离（比如同一个函数可能被多次调用处理不同位置的数据）。另外我们还需要限制这些代码片段的出口和入口（比如函数只能通过调用进入、返回退出，不能跳转到中间开始或者结束时跳转到非调用者代码），达到限制控制流的目的。

所以我们选择代码片段作为安全判定的主体，即所谓“代码即主体”(The code is the subject.)。

动态权限： 基于代码为主体的思想，我们设计安全机制的基本思路是，调用代码片段之前，根据预设或推定的代码数据访问范围，动态设置好允许访问的数据边界和跳转目标。代码片段在允许的数据范围内可以自由访问和跳转，而一旦发生越界访问或者非法跳转，就触发异常阻止访问。代码片段执行返回后，权限自动回收。

动态权限划分需求表现在如下几个方面：

- 需要进行权限的动态设置和分配，因为安全资源或者说安全元数据是有限的，而复杂程序需要对数量众多的代码片段进行隔离和划分。
- 需要支持不同时间下同一段代码片段的不同权限设置，例如函数 A 在第一次是被可信代码调用，这时可以访问私密数据，但是第二次是被非可信代码调用，就需要禁止函数 A 对私密数据的访问。
- 需要应对不能进行静态划分的场景，例如函数指针，调用者在调用之前不知道这个函数的具体行为和含义，就不能提前推定可以访问的数据范围。

内置可信基: 前文基于安全元数据的问题说明，安全资源的管理需要依靠一个可信核心来做。因此我们需要一个可信代码核心（可信基）来统一做权限的管理和元数据的维护，这个可信代码核心需要满足一些基本特性，比如可信基本身需要保证本身不会或者至少很难被攻击，以及可信基到非可信代码需要高效的转移方式以降低性能开销。可信基必须的功能如下：

- 可信基需要有分配和回收安全资源的能力，以支持有限安全资源下对不同代码片段、或者同一代码片段不同时刻的安全防护，以实现动态划分。
- 可信基需要在控制流转移到非可信区之前，设置好非可信区能够访问的数据区域以及进行的操作，保证数据流完整性。
- 可信基需要对自己的入口进行限制，防止非可信区通过控制流劫持跳转到可信区修改安全元数据。
- 可信基需要在非可信代码片段做出违反权限的操作时，触发异常进行处理，可选择退出或者修正。
- 可信基需要提供一系列类似于系统调用的功能，为非可信代码提供本身能访问的数据和功能之外的可信调用。

关于可信基的实现方法，我们没有采用在同一地址空间内部再划分特权级和特权态的方法。我们发现可信基本身也可以作为特殊代码片段进行管理。可信基到非可信基之间的转移不需要进行复杂的特权级切换，而是可以在安全规则控制下直接跳转。因此，可信基实际上也是内置在应用程序中的一个代码片段。

我们把上述基于代码片段做权限动态划分的安全处理器设计方案，称为 Dynamic in-Address-Space Isolation by Code Segments（DASICS）。具体来说，DASICS 方案主要由包含区域划分、代码权限划分设置、控制流限制、权限检查、权限例外处理、主函数调用以及系统调用截获等几个部分组成：

4.1 区域划分

DASICS 以代码片段为单位进行权限划分，具体的代码片段含义是地址空间连续的可执行代码集合（比如一个非 inline 函数内的代码）。代码片段范围可以通过代码起始和结束地址来限定。DASICS 遵循高权限的代码片段可以对低权限的代码片段进行权限限制的原则。如图 3 所示，这种限制是两个维度上的：

- 纵向限制：DASICS 支持高特权级代码对低特权集代码进行权限设置和限制。例如 RISC-V 架构中，M 态的可信区可以对 S 态的可信区进行访问限制，S 态的可信区也可以对 U 态的可信区进行访问限制。
- 横向限制：DASICS 方法将同特权态下的程序代码划分为可信区和非可信区。可信区对非可信区的访问权限进行限制。比如在用户态中，程序的可信区是主函数，而非可信区则是第三方库函数，主函数在调用库函数之前先设置其权限。而在内核态中，可信区是内核调度代码，而像非可信的第三方驱动代码就会被放到非可信区，内核在使用这些非可信驱动时先进行权限设置。

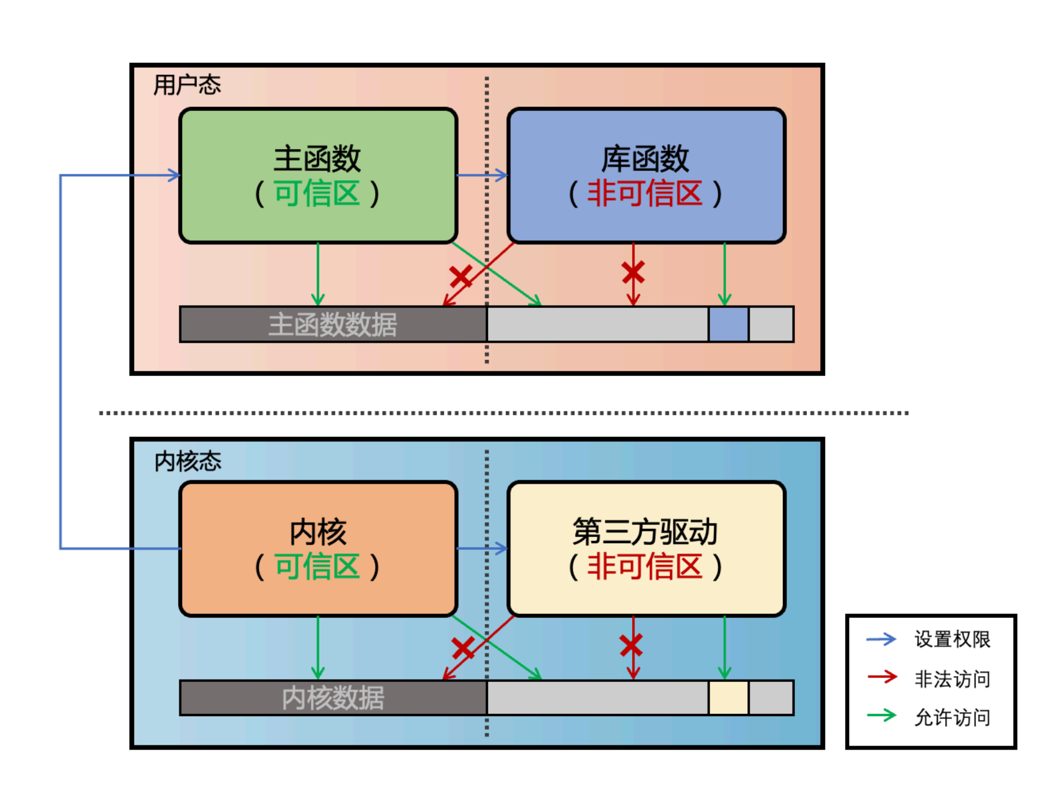


图 3: DASICS 的区域划分（横向限制 + 纵向限制）

可信/非可信区划分目前是通过人工标注地址空间来决定的，DASICS 在链接时会将可信代码片段放到静态指定的地址空间里完成可信区的构建。除此之外其他地址空间的代码则被认为是非可信的。可信区可以访问同一特权级的所有数据，而非可信区不能访问可信区的数据。以此完成不同权限代码的访问区域隔离。

4.2 代码权限划分

在 DASICS 中，代码片段的权限分为访问权限和跳转权限。其中访问权限使用访问数据的边界来划定，DASICS 在硬件中添加了几组边界寄存器，每组寄存器描述了一个可访问区域的范围，以及可以对这个区域进行的操作（读/写）。跳转权限同样也有跳转范围寄存器来规定库函数允许跳转的目标地址范围。

可信区在调用非可信区代码之前，首先分配边界寄存器并设置好非可信区代码的权限，包括访存权限和跳转权限，以此来限定非可信区代码的数据流和控制流。在非可信区返回之后，对边界寄存器资源进行回收。在控制流限制上，除了通过跳转范围寄存器划定非可信区可以访问的区域外，还对非可信区向可信区跳转的入口进行了限定。即非可信区只能通过函数返回或者主函数调用入口和同特权级中断跳转到可信区代码。以此来防止可信区代码被截获利用。

4.3 权限检查

DASICS 的访存权限检查会设置于处理器取指单元和访存单元中。当非可信区的代码发生访存（包括取指/读取/写入）时，使用访存地址到边界寄存器中检查是否访问了允许范围之外的地址、是否对地址空间进行了不允许的访存操作（执行/读/写）。如果发现超越权限的访存操作（地址越界或者无操作权限），则会触发同特权级中断进行进一步的处理（比如报错退出）。

控制流限制的检查会在跳转指令执行时进行检查。主要是检查可信区进行非可信区函数调用的控制流：当程序控制流通过函数调用方式从可信区代码转移到非可信区代码时，在可信区代码里的函数返回地址会被记录下来。等到非可信区代码返回时，对比实际的返回地址和记录里的返回地址，若不一致说明返回地址可能被非可信代码篡改，此时会触发中断报错退出。

基于以上权限检查，DASICS 可以防止恶意代码进行诸如数组越界访问（buffer overflow）以及修改返回地址的控制流劫持（Control-flow hijack）攻击。

4.4 同特权级例外处理

DASICS 支持可信区注册例外处理函数，以处理访问例外或者控制流例外等。根据上述权限检查的描述，当发生例外时候，会触发同特权级的中断（例如在用户态就触发用户态中断）跳转到例外处理函数中，并读取硬件特权寄存器中的例外原因分别进行不同的处理，如报错退出或者动态对权限进行修正后继续执行原指令。例外处理函数也支持动态注册，以应对针对不同场景下统一例外的不同处理。

4.5 可信调用

和操作系统的系统调用的思想类似，DASICS 在限制非可信区的数据访问的同时，也会提供可信调用给非可信区进行可访问范围之外的可信操作。原因是可能存在可信区代码提前设置好的数据范围不足以支撑非可信区代码完成其功能的情况。例如用户态中一个库函数需要增加自己的数据访问权限或者需要更大的数据区域时，就可以通过主函数调用返回到主函数里更新边界寄存器，再返回到库函数进行执行。

为了支持可信调用，可信区需要首先注册可信调用的入口地址以增加合法的可信区入口。非可信区代码可以和系统调用一样设置好可信调用的参数，然后直接跳转到这个入口进行可信调用。后面这一点比系统调用更加快速，因为并不需要耗时间的特权级切换。

4.6 系统调用截获

为了防止利用跨特权级调用进行的攻击，DASICS 中加入了对系统调用的截获。硬件会设置观测逻辑，一旦捕捉到系统调用指令（例如 RISC-V 架构中的 ecall 指令），就触发用户态中断跳转到可信区例外处理函数。例外处理函数根据例外原因（用户态系统调用例外）再跳转到系统调用例外处理函数，这个函数中可以针对本次系统调用进行一些安全检查，比如检查这个代码片段是否能进行这个系统调用、检查系统调用的参数是否越过了这个代码片段的权限范围以及对一些非安全的系统调用进行代理（复制系统调用的参数之后由主函数来代理进行系统调用）。在应用内部进行系统调用检查还可以更有针对性的对各种文件、设备资源进行细粒度的管理。

4.7 DASICS 方案的优势

相较于之前针对内存访问漏洞的安全保护机制，我们认为 DASICS 方案的优势在于：

- **提供全面的保护**：DASICS 通过不同权限主体的内存隔离来保证数据流完整性（DFI），通过函数调用和返回的配对检查来保证控制流完整性（CFI）。同时提供了数据流和控制流两个方面的保护并互相支持，除了对应用数据提供保护外，安全元数据本身被修改的风险也很小。
- **提供高效率保护**：DASICS 的可信区到非可信区之间的转移是直接的跳转，不需要像 Donky 等保护机制通过中断的方式进行切换，更可以避免频繁进出内核、管理态等来实现完整的安全保护机制。
- **提供细粒度的保护**：DASICS 通过动态基于边界寄存器的访问范围划分来达到对指针的访问限制，相较于基于页粒度的保护机制（如 MPK）提供了更精细的保护粒度。DASICS 的代码片段，不依赖某种硬件设置的 id 或 tag 等进行区分，可以进行更细粒度的权限划分。
- **提供动态的保护**：DASICS 通过可信区的动态设置实现了区域的动态划分，通过安全资源（访问边界寄存器和跳转范围寄存器）的分配和回收实现了权限的动态划分。实现了不同场景下根据需求进行的动态、灵活的保护，可以支持具有时效性的安全规则。
- **提供多特权级的保护**：DASICS 支持多级特权态的安全防护（内核态、用户态），不仅可以用于用户态代码的保护（见 6.1 Use Case 1），还可以进行内核代码的防护（见 6.2 Use Case 2）
- **有很好的可移植性**：DASICS 保护机制下，用户只需要将第三方库划分到安全区，而不需要对第三方库进行重新编译，降低了第三方库代码的重编译困难。在可以修改源码的情况下，只需要在函数出入口增加简单的 API，就可以实现进一步的安全增强。

5 原型 1.0 的实现

我们基于上述设计思路实现了 DASICS 的原型 1.0，包括硬件和软件原型。硬件部分基于国内开源的 RISC-V 处理器 NutShell 实现，软件部分则是基于 BBL(Berkeley BootLoader) 与 Linux 4.18.2 进行了修改。

5.1 硬件实现

DASICS 硬件部分基于开源 RISC-V 架构处理器 NutShell 实现。在 NutShell 顺序核的六级流水线结构中，我们添加了对 DASICS CSR、DASICS 例外和 DASICS 新指令的支持，涉及到的模块主要包括译码模块（IDU）、CSR 执行模块和访存模块（LSU）等，总共改动的 chisel 代码行数不超过 500 行。

对于 DASICS CSR 而言，我们在 CSR 模块中新增了如下表所示一共 43 个位宽为 64 bits 的逻辑 CSR 寄存器，用来支持 DASICS 的安全防护功能。

DASICS CSR 名称	CSR 编号	功能
dsmcfg & dumcfg	0xBC0 & 0x5C0	S 态/U 态主函数配置寄存器
dsmbound0 & dsmbound1	0xBC1 & 0xBC2	S 态主函数边界寄存器
dumbound0 & dumbound1	0x5C1 & 0x5C2	U 态主函数边界寄存器
dlcfg0 & dlcfg1	0x881 & 0x882	库函数配置寄存器
dlbound0 ... dlbound31	0x883 ... 0x8A2	库函数边界寄存器
dmaincall	0x8A3	主函数调用入口地址
dretpc	0x8A4	库函数返回地址
dretpcfz	0x8A5	活跃区返回地址

- dsmcfg 与 dumcfg 共用一个物理 CSR 寄存器，用于使能 DASICS 对于 S 态/U 态的保护，并且设置了 S 态/U 态的清零位，以降低软件切换上下文时的开销。
- dsmbound0/1 和 dumbound0/1 分别划定了 S 态和 U 态的可信区边界，不在这个边界范围内的都为非可信区。
- dlcfg 为库函数配置寄存器，每个 dlcfg 内都包含着 8 个 4 位的 **tiny config**（奇数索引的 tiny config 为全零不使用），而每个 tiny config 都对应着一组 dlbound，表示这组库函数边界内的代码片段具有哪些权限（读/写/可执行）。dlcfg 与 dlbound 一起组成了一个 DASICS 库函数查找表，用于判断是否发生了 DASICS 例外。
- dmaincall 存储主函数调用的入口地址，库函数跳转到该地址的行为是被允许的。
- dretpc 记录了库函数调用完成后，允许其返回的目标地址。当主函数使用跳转指令或者分支指令跳转到库函数时，跳转指令的下一条指令的地址将会被硬件自动填入 dretpc 中。
- dretpcfz 为活跃区返回地址，所谓活跃区是指非可信区代码中，外部允许跳转到的内部任意代码，但是该内部的代码只能通过函数返回跳转到外部的代码区域。这个机制是对嵌套的非可信区调用所做出的进一步限制

为了防止非可信区内的代码通过修改上述 CSR 寄存器来使 DASICS 保护功能失效，DASICS 机制也为读写 CSR 寄存器设置了新的限制，即仅允许可信区内的代码访问特权态允许的 DASICS CSR 寄存器。例如对于 dsmbound0/1 而言，仅允许 M 态、S 态主函数对其进行读写；对于 dlcfg 而言，则仅允许 M 态、S 态主函数、U 态主函数对其进行读写。

为了实现发现 DASICS 访问或者跳转违例时触发例外的功能，以及进行系统调用劫持，我们新增了如下表所示的 8 种 DASICS 例外：

- 对于 DASICS S 态/U 态执行例外而言，当非可信区函数跳转到可信区的跳转目标地址不等于 dretpc 或者 dmaincall 时，或是非可信活跃区函数想要跳转到返回地址之外的地址时，亦或是非可信区代码跳转到边界寄存器所允许的跳转范围时，引发该例外。

DASICS 例外名	例外号	描述
DASICS_U/S_INST_FAULT	0x18 / 0x19	U 态/S 态跳转例外
DASICS_U/S_LOAD_FAULT	0x1a / 0x1b	U 态/S 态无读权限
DASICS_U/S_STORE_FAULT	0x1c / 0x1d	U 态/S 态无写权限
DASICS_U/S_ECALL_FAULT	0x1e / 0x1f	U 态/S 态系统调用例外

- 对于 DASICS S 态/U 态访存例外而言，当非可信区试图访问允许的访存范围外的地址，即访存地址与读写标记在 DASICS 库函数查找表中没有匹配项时，则根据当前指令是 load 还是 store 分别引发对应特权态下的 DASICS 读/写访存例外。同时，当该例外发生时，需要取消掉 LSU 模块中正要发送给 AXI 总线的访存请求。
- 对于 DASICS S 态/U 态系统调用例外而言，如果发现当前指令是一条 RISC-V 系统调用指令（即 ecall 指令）就触发该例外，在例外函数中进行系统调用的截获检查。

在 NutShell 的指令译码模块中，我们添加了 dasicsret 这一新增指令的译码逻辑。该指令的设计参考了 JALR 指令，但与之不同的一点在于，dasicsret 指令仅允许主函数调用。

此外，为了加速软件对 DASICS 例外的处理流程，我们在 NutShell 上按照 RISC-V N 扩展的草案（版本 1.1）实现了 RISC-V 架构的 N 扩展，从而可以通过配置 medeleg/sedeleg 将 DASICS 例外代理到用户态进行处理。当 DASICS 例外被代理到用户态之后，其例外处理流程如图所示。

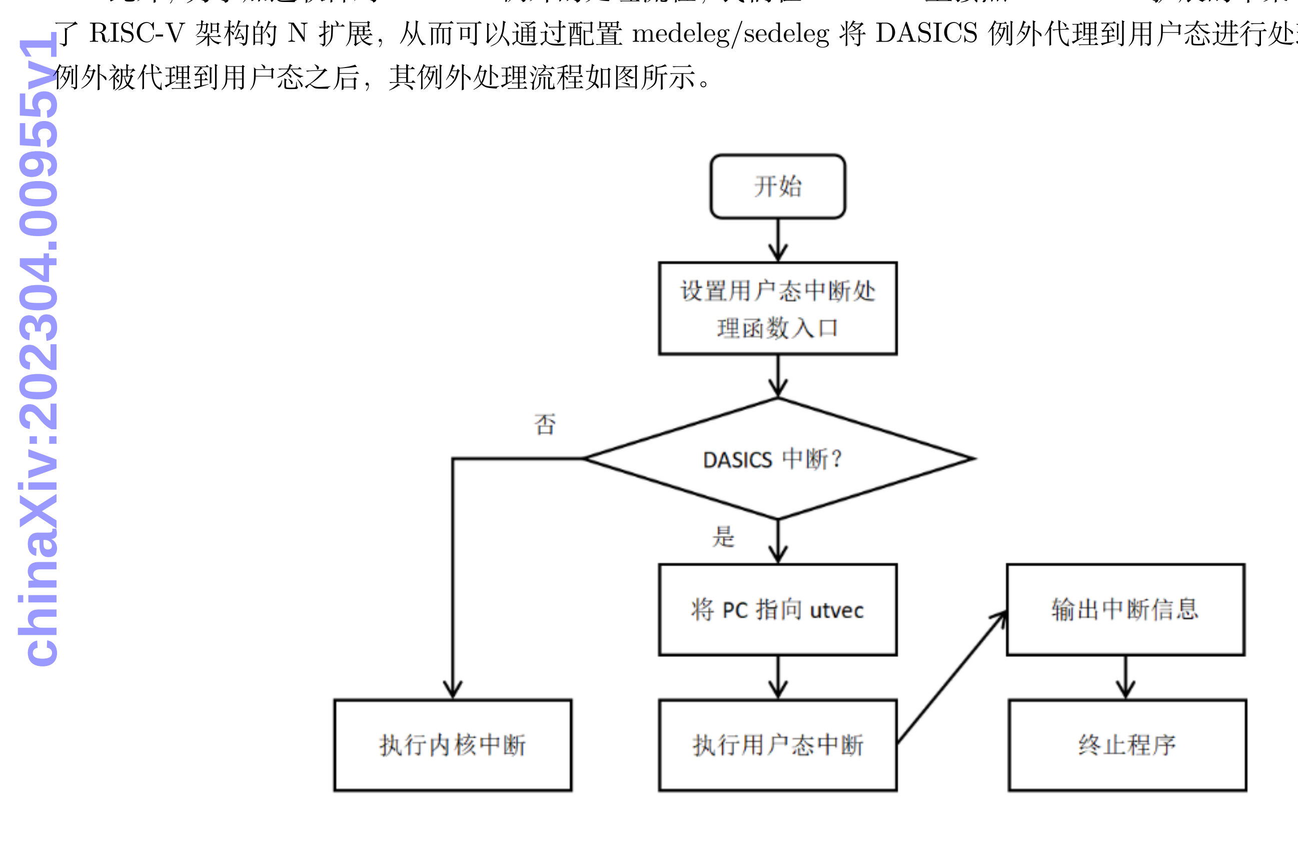


图 4: DASICS 的例外处理流程

5.2 软件实现

在 BBL 中，我们增加了 DASICS 机制的初始化，包括设置 DASICS 例外的代理，以及设置 dsmbound 为全内存空间。后者这样做的原因是：（1）BBL 无法确定 S 态可信区的大小；（2）BBL 完成初始化之后会跳转到 S 态可信区中，由 S 态可信区通过新增的 SBI 调用来修改 dsmbound。同时，我们也在 BBL 中增加了修改 dsmbound 的 SBI 调用，该 SBI 调用仅能由 S 态可信区调用，否则会直接报错返回。

在 Linux 中，我们增加了 DASICS 例外处理的逻辑，并且将 DASICS 机制运用到用户程序的 ELF 加载中。对于前者而言，我们在 linux 原本的例外处理流程中增加了保存恢复 DASICS 程序上下文的逻辑，并且将 DASICS U 态例外代理到用户态处理。与此同时，我们在 linux 中提供了默认的主函数调用入口函数，以及默认的 DASICS 用户态例

外处理函数。对于用户程序的 ELF 加载而言，我们主要修改了 fself.c，通过识别用户程序中的 dasics 分区，从而填写 dumbound 寄存器以设置用户程序的主函数区域。

6 DASICS 应用场景举例

本章介绍几个可能的通过 DASICS 高效的提高系统安全性的应用实例。

6.1 Use Case 1: 可信区 Rust + 非可信 C 代码库

前文已经提到，通过以 Rust 为代表的安全编程语言进行编程可以解决大部分的内存漏洞，但是这类安全编程语言开发复杂，移植困难，将现有的所有程序代码都用安全语言进行改写会带来巨大的工作量。加入 DASICS 之后，开发人员可以使用安全语言构建可信区代码，然后其他的非可信库代码使用 DASICS 机制进行限制。这样既保证了可信区代码本身不会出现内存漏洞，也大大降低了改写的工作量。非可信区的代码权限通过 DASICS 机制在可信区进行了限制，从而大大降低了非可信区中可能存在的安全漏洞带来的危害。DASICS 机制通过对硬件寄存器的操作实现对安全权限的设置，与编程语言无关，因此可以支持各种编程语言，并提供灵活的安全设置。

6.2 Use Case 2: 内核保护 + 第三方驱动

现有的许多安全手段都依赖一个安全的内核，通过在内核中设置某些安全权限数据，从而实现对用户态程序的安全检查或限制。但是内核态本身可能也运行着大量的第三方代码，包括文件系统、设备驱动、网络协议栈等内核模块。这些模块本身对于操作系统的开发者来说并不属于可信任的代码，从而存在安全隐患。第三方模块和内核代码都运行在特权级，一旦操作系统内核本身的安全性被破坏，当前大量的安全防护手段都将失效。通过 DASICS 机制，我们可以在操作系统中也加入可信区与非可信区的隔离。系统开发者可以将第三方的模块代码都加入内核空间的非可信区，并限制它们的执行访问权限，从而避免第三方内核模块带来的操作系统安全风险。DASICS 机制通过硬件实现安全保护机制，从而可以支持内核地址的隔离与保护，既不会破坏原本的内核态与用户态的隔离，又在内核态增加了新的隔离机制。

6.3 Use Case 3: single level OS （LibraryOS）

在云计算等场景中，应用程序往往不需要复杂的操作系统功能，而复杂的操作系统又会降低整个系统运行的效率。因此，LibraryOS 或 Unikernel 的概念被提出，通过将应用程序需要的操作系统功能变成用户态库的方式给应用程序使用，从而减少了复杂操作系统带来的开销，且提高了应用程序的平台适配度。但是这样的方案获得性能的同时失去了传统操作系统的安全性，即内核态与用户态的隔离。为了达到足够的安全性，又必须再引入额外的手段实现轻量级的隔离机制。如果 LibraryOS 加入 DASICS 功能之后，可以限制用户程序的访存权限，并且可以以函数为粒度进行限制，从而隔离 LibraryOS 中应用程序与内核功能的代码，避免一定的安全隐患。相比于基于多个保护态切换的保护机制，DASICS 的性能开销更小，从而可以保持 LibraryOS 原本的性能优势不被破坏。